10 Most(ly dead) Influential Programming Languages

www.hillelwayne.com/post/influential-dead-languages/

March 25, 2020

The other day I read 20 most significant programming languages in history, a "preposterous table I just made up." He certainly got preposterous right: he lists Go as "most significant" but not ALGOL, Smalltalk, or ML. He also leaves off Pascal because it's "mostly dead". Preposterous! That defeats the whole point of what "significant in history" means.

So let's talk about some "mostly dead" languages and why they matter so much.

Disclaimer: Yeah not all of these are dead and not all of these are forgotten. Like most people have heard of Smalltalk, right? Also there's probably like a billion mistakes in this, because when you're doing a survey of 60 years of computing history you're gonna get some things wrong. Feel free to yell at me if you see anything!

Disclaimer 2: Yeah I know some of these are "first to invent" and others are "first to popularize". History is complicated!

Detecting Influence

Before we start, a quick primer on finding influence. Just knowing that X was the first language with feature Z doesn't mean that X actually *influenced* Z. While Absys was arguably the first logic programming language, almost all of logic programming actually stems from Prolog, which was developed independently. Ultimately there's only one way to know for certain that X influenced Y: citation. This means one of

- Y cites X in its reference manual
- Y cites a paper that cites X
- The author of Y says "we were influenced by X."

Citations are transitive. Sometimes the language manual for Q lists motivating document R, which cites paper S as an inspiration, which mentions it got the ideas from language T. Then we know that T influenced Q, even if the chain is several steps long. This means digging through many sources to find a signal. To speed this up we use heuristics to decide where to look.

One effective heuristic is programming language **cognates**. It's very rare for languages to independently come up with the same syntax. So if two languages share some syntax, one likely influenced the other. For example: even without reading design decisions by Matz, we know that Ruby was influenced by Smalltalk, as they both filter a list with a `select` method. This isn't conclusive evidence. Maybe Matz came up with it independently, or maybe Ruby and Smalltalk were both influenced by a common ancestor. But it gives us a place to start looking.

The Languages

COBOL

Background: CODASYL, 1960. COBOL is shaped by the business/science split in computing. At that time high-level industry languages were either used for engineering computations or managing data. The engineers had all gotten behind FORTRAN while the business world was a mess of COMTRAN, FLOW-MATIC, and others, so the Department of Defense got a committee together to make a single universal business language. That's COBOL.

COBOL was one of the four "mother" languages, along with ALGOL, FORTRAN, and LISP. While we consider it a punchline today, it was once the most popular language in the world. It still runs a lot of our legacy business systems.

Significance: In terms of syntax and semantics we don't see much of COBOL in modern computing. COBOL's most important addition is the concept of record data. In FORTRAN and ALGOL, your only data structure was the static array. In COBOL, though, you could read in structured files with hierarchical data, and it would automatically destructure them into the representative variables. This was a precursor to modern day structs.

Cause of Death: Two factors here. One: COBOL had no overlap with other PLT efforts. Very few people built on COBOL. This meant that second or third generation languages, which built on the lessons of their ancestors, had almost no COBOL DNA. This was less intrinsic problem of COBOL and more because of the academia's disdain for its creation process. CODASYL was a business group and obviously wasn't worth paying attention to.[1] COBOL was also enormously complex, even for today's languages. This means that COBOL compilers lagged contemporaries on microcomputers and minicomputers, giving spaces for other languages to flourish and eventually outcompete it.

ALGOL

Background: The ALGOL committee, 1960. ALGOL-58 came out two years before but was quickly superseded, so I'm wrapping them into each other. The committee wanted to make a good language for researching algorithms. In other words, ALGOL was a formalized "pseudocode".

Of the four mother languages, ALGOL is the most "dead"; Everybody still knows about LISP,[2] COBOL still powers tons of legacy systems, and most scientific packages still have some FORTRAN. But I've met plenty of programmers who haven't even heard of ALGOL. You'd think it'd be the least important of the mother languages, but it's the opposite. Of the four, only LISP comes anywhere *close* to the pervasive importance of ALGOL.

Significance: Let's see: lexical scoping, structured programming, nested functions, formal language specifications, call-by-name semantics, BNF grammars, block comments… every modern language today is deeply influenced by ALGOL.

Cause of Death: ALGOL was a research language, not a commercial language. It was designed for studying algorithms. The spec didn’t define any I/O, which kinda made it impossible to use in practice. Sure, you could write a compiler extension, but then you might as well add other stuff too.

And that’s exactly what people did. In 1960 and 70 people made a huge number of ALGOL-likes by extending ALGOL with I/O and extra data structures. This includes JOVIAL, SIMULA, CLU, and CPL. Later languages were then based off these extensions, not ALGOL directly. We call C an “ALGOL-like”, but it’s actually a BCPL-like, which was a CPL-like, which was an ALGOL-like. ALGOL’s children buried it.

Eventually the ALGOL people tried to extend it into ALGOL-68, which radically departed from ALGOL-60 and hasn’t had close to the same influence. The ALGOL-60 lineage continues with Niklaus Wirth’s Pascal.

APL

Background: Ken Iverson, 1962. Originally a hand-written notation for array math, IBM picked it up and used as an programming language. As a language, APL focused on array processing: being able to concisely manipulate large blocks of numbers.

If you’ve heard of APL before, you probably know it as “that weird symbol language”. One of the most notorious code snippets is this implementation of the Game of Life:

```
life←{↑1 ω∨.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂ω}
```

You had to write it with a specialized keyboard, like this:

An APL keyboard
(source)

Nonetheless, APL got popular on mainframes for running with very low memory requirements.

Significance: Array processing. At a time when adding two lists of numbers meant a map or a loop, APL introduced the idea of operating on the entire array *at once*. For example:

```
   1 + 1 2 3 4
2 3 4 5
   1 2 3 4 + 1 2 3 4
2 4 6 8
   2 4 ⍴ ⍳8
1 2 3 4
5 6 7 8
   1 2 3 4 +[2] 2 4 ⍴ ⍳8
2 4  6  8
6 8 10 12
```

This was a *really big deal* in scientific circles. So much applied math boils down to large-scale operations on large matrices. When you can just take the outer product with `∘.f`, it's really damn easy to take outer products!

Through this innovation APL lead to R, numpy, pandas, Matlab, etc. There's also the direct descendants of APL: J, Dyalog, K, Q. They've been less successful but still see lots of use in the finance sector.

Cause of Death: Well, the obvious problem is the keyboards. If you can't write it in ASCII, you're not going to write very much of it. Iverson fixed this with J, which uses digraphs instead of different symbols. Instead of `≠`, you write `~:`. This was in 1990, though, which is a bit late to popularize a radically different programming style.

The subtler problem is that APL and J only worked on homogeneous data. You can't store both strings and numbers in the same data structure (unless you use boxes, which is a whole other can of worms) and working with strings is generally a nightmare. So no dataframes, which excludes a lot of modern data science.

Further Reading: Notation as a Tool of Thought

BASIC

Background: John Kemeny, 1964. Originally a simplified FORTRAN-like, intended to help people outside engineering use computers.

BASIC really took off in the microcomputer era. The first microcomputers didn't have enough memory to compile "real" programming languages, whereas you could cram a pared-down BASIC compiler into like 2 kilobytes. BASIC became a *lingua franca* for early-stage programmers. If you were programming at home in the 1970's, you were probably writing BASIC on a microcomputer.

```
10 PRINT "Hello, World!"
20 END
```

Significance: The biggest *technical* impact is runtime interpretation. BASIC was the first language with a real-time interpreter (the Dartmouth Time Sharing System), beating APL by a year. And that APL system was only available to IBM customers, so really it was BASIC or nothing for a long time.[3]

BASIC had a bigger *social* impact. It brought programming to households, kids especially. Many of the influential programmers in the 80's and 90's first learned how to program on BASIC. Many enterprise programs were also written in BASIC, which probably helped accelerate the decline of Cobol.

BASIC has one more neat trick up its sleeve: Office tooling! Microsoft eventually turned BASIC into Visual Basic, which they used as the Office macro language. This then spread to OpenOffice and LibreOffice, entrenching BASIC in that particular niche. More recently it's lost ground to JavaScript and is now a legacy macro language.

Cause of Death: People saw BASIC as a “lesser” language. You might use it if you were a kid or a small business owner, but *real* programmers used a *real* language. Once manufacturers could cheaply make microcomputers with more than 16k of RAM they started depreciating BASIC for languages like Pascal and C.

BASIC lived on for a while as a legacy kids teaching language, but seems to have died out of that niche, too.

PL/I

Background: IBM, 1966. IBM’s business was split into two languages: FORTRAN for scientists and COMTRAN for business folk. Facing competition from COBOL and wanting to streamline their systems, they tried to make a language that was useful for both engineering and business purposes. This ended up looking like a sort of superset of the two languages, with a bunch of additional features stapled on top.[4] Now everybody could use the same language and IBM can make a lot more money! Yaaaaaaaay

Significance: The authors of ALGOL-68 mockingly called PL/I an obsolete language. But everything ALGOL-68 did, PL/I did earlier and better. While COBOL got structured data first, PL/I was the first language to implement them as a type. In COBOL, reading in a user with a name would give you two global variables, `user` and `name`. In PL/I, you’d get one variable with a field, `user.name`. PL/I was also the first high-level language with pointers for direct memory manipulation, constants, and function overloading.[5]

Many of these ideas entered mainstream programming via C, which was a mix of both BCPL and PL/I. C even uses PL/I’s comment syntax.

Cause of Death: All the FORTRAN programmers thought it was too much like COBOL and all the COBOL programmers thought it was too much like FORTRAN. IBM had tried to take on two established languages with a much more complicated one. It didn’t help that they were the only group with the compiler, meaning everybody else was mistrustful of vendor lock-in. By the time IBM was able to make headway in both of these issues the wider computing world had already moved on to the microcomputer era, where PL/I was out competed by BASIC.

Further Reading: The Choice of PL/I

SIMULA 67

Background: Ole Dahl and Kristen Nygaard, 1967. They extended ALGOL for doing simulations. First they made SIMULA I, which had dedicated simulation and “activity” syntax. SIMULA I saw some early use, but the two were dissatisfied with how “specialized” the language felt and how much duplicate code they had in their simulations. They wanted to make a more general framework for representing things in general, not simulations only.

Their idea was to allow users to define new types called “classes” with polymorphic function resolution. Then users could build the simulation features as a special case of the object system, making it easy to customize how it all worked to their particular needs.

Significance: While SIMULA wasn’t the first “true” OOP language, it was the first language with proper objects and laid much of the groundwork that others would build on. This includes the class/object split, subclassing, virtual methods, and protected attributes. It inspired almost all of the academic research into objects after 1967. Both CLU and ML cited SIMULA as a major source of inspiration. Bjarne Stroustrup did his PhD on SIMULA, eventually incorporating a lot of its ideas into C++.

Cause of Death: In that same PhD Stroustrup claimed that SIMULA was waaaaaay too slow to use at scale. “Good luck getting anything done if you aren’t on a mainframe” slow. It’s worth noting that Smalltalk-80, which took the same ideas even further, had an extra 13 years of Moore’s law behind it. And even Smalltalk was often mocked as too slow. Everybody went and implemented the ideas in SIMULA that they could integrate into faster, simpler languages.

Further Reading: Compiling SIMULA: a historical study of technological genesis, The History of Simula

Pascal

Background: Niklaus Wirth, 1970. Made to capture the essence of ALGOL-60 after ALGOL-68 got waaaaaay too complicated for Wirth’s liking. It first got big as the “introduction to CS” language, and by the early 80’s was the second-most popular language on the Usenet job boards. Wirth considers the whole family- Pascal, Modula, and Oberon- as a single unified language concept.

Significance: Pascal didn’t introduce any completely new ideas. It was an intentionally conservative language that tried to pick the best parts of the past decade and provide them in a unified package. Pascal brought ALGOL syntax outside academia, so much so that ALGOL’s assignment syntax, `:=`, got called “Pascal style” instead. From this point on most language features that look like ALGOL were more likely inspired by Pascal than directly by ALGOL itself.

While Pascal wasn’t very innovative, variants of it were. Wirth also pioneered the idea of “stepwise refinement” as a means of writing rigorous software. This eventually lead to the Modulas, which popularized first class software modules, and Euclid, the first formal verification language to see production use.

Cause of Death: I’m calling a mulligan on this one. Unlike most of the other ones on this list, Pascal didn’t have major structural barriers or a sharp competitor. Sure, it competed with C, but it was still doing fine for a very long time. People usually attribute the Why Pascal is not my favorite language essay, but that’s too neat of an answer and history is a lot messier. Also, Delphi is still pretty high-ranked in the TIOBE and PYPA measurements, so it's not exactly *dead* in the same way SIMULA is. An accurate analysis of the fall of Pascal would be longer than the rest of this essay.

Further Reading: The Programming Language Pascal, Pascal and its Successors

CLU

Background: Barbara Liskov, 1975. Liskov wanted to mess around with abstract data types. That's it. That's the whole reason for CLU.

Significance: CLU might be the most influential language that nobody's ever heard of. Iterators? CLU. Abstract data types? CLU. Generics? CLU. Checked exceptions? CLU.

We didn't adopt the same terminology, so it's not 100% obvious it all comes from CLU, but still. Every language spec for the next decade would namedrop CLU. CLU did a lot.

Cause of Death: CLU was a showcase language; Liskov wanted to get people to adopt her *ideas*, not her specific *language*. And they did: almost every language today owes *something* to CLU. As soon as she completed CLU she moved on to **Argus**, which was supposed to showcase her ideas on concurrency. That hasn't seen nearly the same adoption, and there's still a lot of stuff in it left to mine.

Further reading: A History of CLU

ML

Background: Robin Milner, 1976.[6] Milner was building the LCF Prover, one of the first **proof assistants.** If you wrote a proof in the right format, LCF could check to see if it was correct or not. To assist in writing the proofs, Milner created a *metalanguage* based on sound mathematical formalisms, which at the time meant strict static types and higher-order functions. Eventually ML was standardized as Standard ML.

Significance: ML is arguably the oldest "algebraic programming language". There's a lot of stuff we attribute to ML: algebraic data types, modules, typed functional programming. Surprisingly, it was *not* the first for a lot of these! The first ML was just designed to work with LCF and wasn't a general purpose language, so lacked a lot of these features. As people started making it more general they pulled ideas from other research languages and incorporated them into ML. One *very* important idea did start in ML, though: type inference. ML was the first statically-typed language where you didn't *have* to write the types out, as the compiler would figure out the types for you. This paved the way for typed FP to escape academia and enter production use.

ML also greatly influenced modern theorem provers. The "program" languages for Isabelle, CVC3, and Coq are ML-based. And a *lot* of type theory was based on ML, though in more recent years the Haskell branch of FP has become more popular.[7]

Cause of Death: ML had a lot of interesting features, but people paid attention to it for the type inference. At the time ML was still a special purpose language for the theorem provers. SML came out the same year as Haskell, which was a much “purer” example of a typed FP language.[6]

Smalltalk

Background: Alan Kay, 1972, 1976, and 1980. It’s sort of a moving target. Smalltalk-72 was the first, Smalltalk-76 introduced the idea of “object-oriented programming” to the wider world, and Smalltalk-80 was the one that saw widespread adoption.

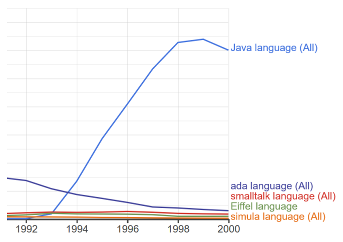
Smalltalk wasn’t the first language with objects but it was the first “object-oriented” one. The difference was that Simula *had* objects in addition to primitives like numbers and booleans, while in Smalltalk, booleans *were also* objects. I wrote a bit about this here if you want to learn more.

Significance: We sometimes think that Smalltalk is “true” OOP and things like Java and Python aren’t “real” OOP, but that’s not true. OOP is a giant mess of many different influences, just like every other paradigm. But it was certainly the thing that *popularized* the idea. If you crack open any general theory OOP book from the mid-80’s or early 90’s, they’ll be in Smalltalk. Many will also translate their examples to C++, and a few will use another language, but everybody will use Smalltalk.

Smalltalk also spread the idea of objects as shareable data, leading the way to CORBA, and it inspired the computational Actor model.

Cause of Death: The common belief is that Smalltalk lost because people used C++ instead. But that’s untrue. Smalltalk did have some issues, specifically its difficulty interoping with other tools and poor runtime performance. But even into the 1990’s Smalltalk was doing respectable business and many people assumed it would be a dominant business language.

Then Java happened.



(source)

Smalltalk wasn’t the only casualty of the “Javapocalypse”: Java also marginalized Eiffel, Ada95, and pretty much everything else in the OOP world. The interesting question isn’t “Why did Smalltalk die”, it’s “Why did C++ survive”. I think it’s because C++ had better C interop so was easier to extend into legacy systems.

This is just a small sample of the important dead languages. I didn't cover ALPHARD, ALTRAN, Argus, Automath, BCPL, COMTRAN, CPL, Eiffel, FLOW-MATIC, HOPE, Hypercard, ISWIM, JOVIAL, MacSyma, Mesa, Miranda, Multics Shell, PLANNER, SMP, Sketchpad, or SNOBOL. All of them contributed in their own way to the modern programming world. History is complicated.

Most influential languages never went mainstream. Few people used any one of them. But each one inspired people, who inspired other people, so the DNA of these forgotten languages appear decades after they're forgotten. But there are also untold languages that didn't get their ideas out. The Encyclopaedia of Programming Languages lists over 8,000 programming languages. Many of them had ideas that never left their bubble. Consider how much we'd have lost if nobody had heard of SIMULA, or Liskov never shared CLU.

That's one reason I love studying history. To learn what we've lost and find it again.

The first draft of this was originally shared on my newsletter. If you found this interesting, why not subscribe?

Thanks to Miikka Koskinen, Kevlin Henney, Eric Fischer, and Levi Pearson for corrections and feedback.

---

1. This is sarcasm. [return]
2. People know about Lisplikes, anyway. Close enough. [return]
3. It's debatable whether BASIC or JOSS was the first, as they both were developed at roughly the same time and both were rolled out in stages. But JOSS only had a few hundred users and had almost no influence on modern languages. [return]
4. For the record, PL/I is pronounced “Programming Language One”, not “Programming Language Eye”. The I is supposed to be the Roman Numeral. [return]
5. PL/I actually called these “generic functions”, but the implementation is identical to overloading. PL/I didn't have any concept of an abstract data type, which was almost a decade away. [return]
6. Several people responded that ML is still alive through F# and OCaml. I was thinking of those as distinct descendents of ML in the same way that Visual Basic is distinct from BASIC. But there's not a clear-cut division here. [return]
7. Haskell draws much more from HOPE and Miranda than it ever did ML. [return]